UNIT 4
Testing methods

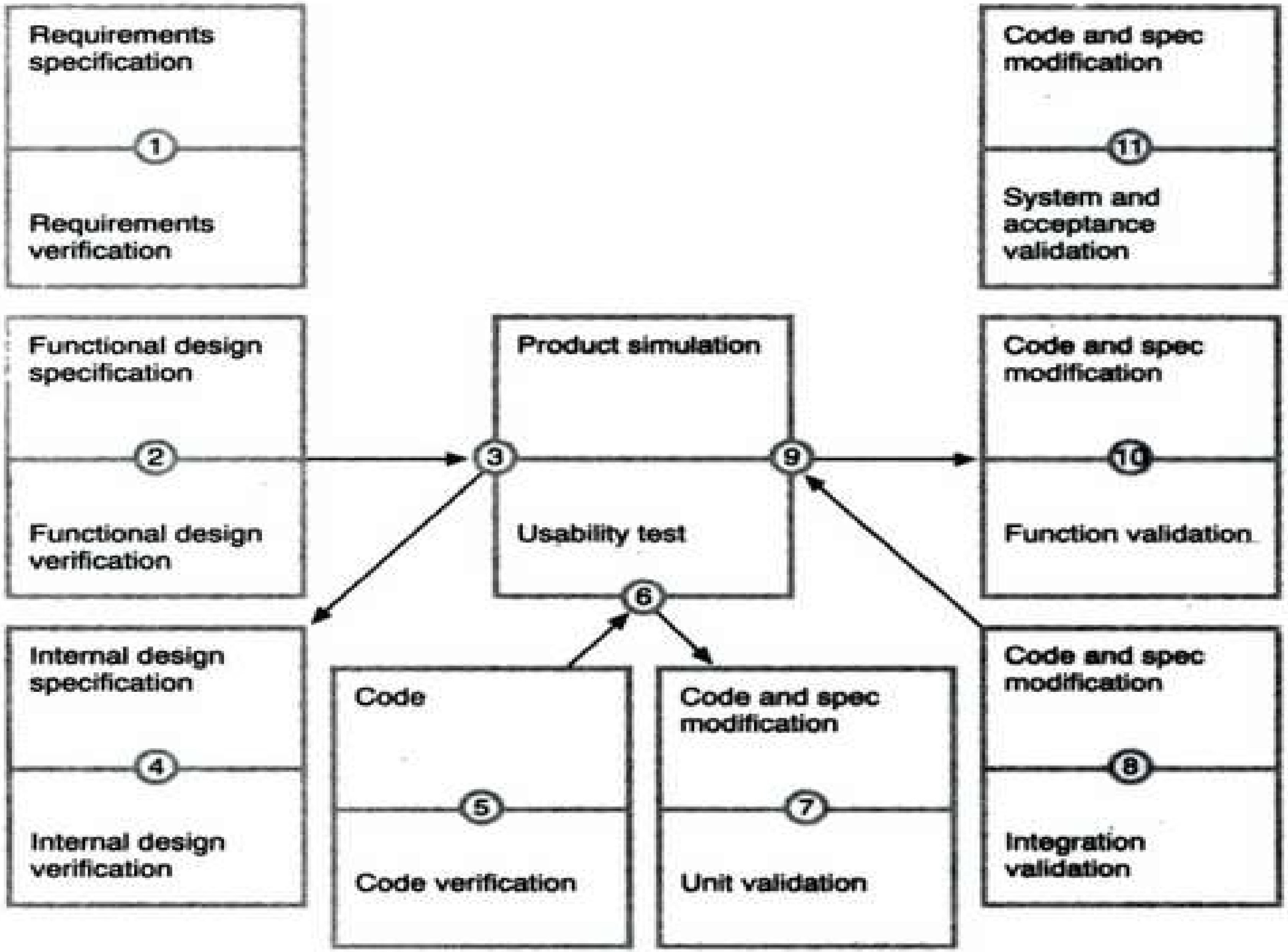
- Verification testing
- Basic verification methods,
- Getting leverage on verification,
- Verifying documents at different phases, getting the best from verification,
- Three critical success factors for implementing verification,
- Recommendation.

Verification testing

- Main objective of each verification activity is
  - to detect as many errors as possible.
  - Testing team need to participate in inspections and walkthroughs
  - Develop their own Testware- generic and test specific checklists and also verify Testware.

Basic Verification Methods

- **“Human”** examination or review of work product.
- Reviews such as inspections, walkthroughs, technical reviews and other methods



Requirements specification
1
Requirements verification
Functional design specification
2
Functional design verification
Internal design specification
4
Internal design verification
Product simulation
3
9
Usability test
6
Code
5
Code verification
Code and spec modification
7
Unit validation
Code and spec modification
11
System and acceptance validation
Code and spec modification
10
Function validation
Code and spec modification
8
Integration validation

Formal structured types of verification

- Inspections: central event
- Meeting at which defects are detected in the product under review.
- Key elements
  - Every one in the review group participates openly and actively.
  - Written report is produced regarding the status of product.
  - Review group is responsible for the quality of information in report.

Inspections : key elements and phases

- objectives:
  - Obtain defects and collect data
  - Meeting does not examine alternate solution
  - To communicate important work product information.
- Elements:
  - Planned structured meeting required individual preparation by all participants
  - Team of 3-6 people, led by impartial moderator
  - Presenter is reader other than producer

- Input:
  - ❑Document to be inspected
  - ❑Related source documents
  - ❑General and tailored checklists
- Output:
  - – Inspection summary/report
  - – Data on error types
- key phrases:
- Briefing/entry/-individual preparation inspection meeting-edit/rework/ exit/re-inspect collect data.
- Casual analysis meeting.

- Walkthroughs
  - Less formal than inspections
  - Lack of preparation
  - Participants simply come to the meeting
  - Presenter prepares
  - No additional effort by participants prior to the meeting.

Walkthroughs Key elements:

- Objectives: to detect defects and to become familiar with the material.
- Elements:
  - A planned meeting where only the presenter must prepare.
  - A team of 2-7 people led by the producer/author.
  - The presenter is usually the producer.

- input:
- Elements under examination
- Objectives for the walkthrough, applicable standards.
- Output:
  - Report

- Cover more material than inspections.
- Purpose for communication.
- Don’t know whats there exactly
- If defects found that’s fine.
- Main goal is to familiarize with product.
- Dis: review tends to be less objective when presenter is producer.

Buddy checks

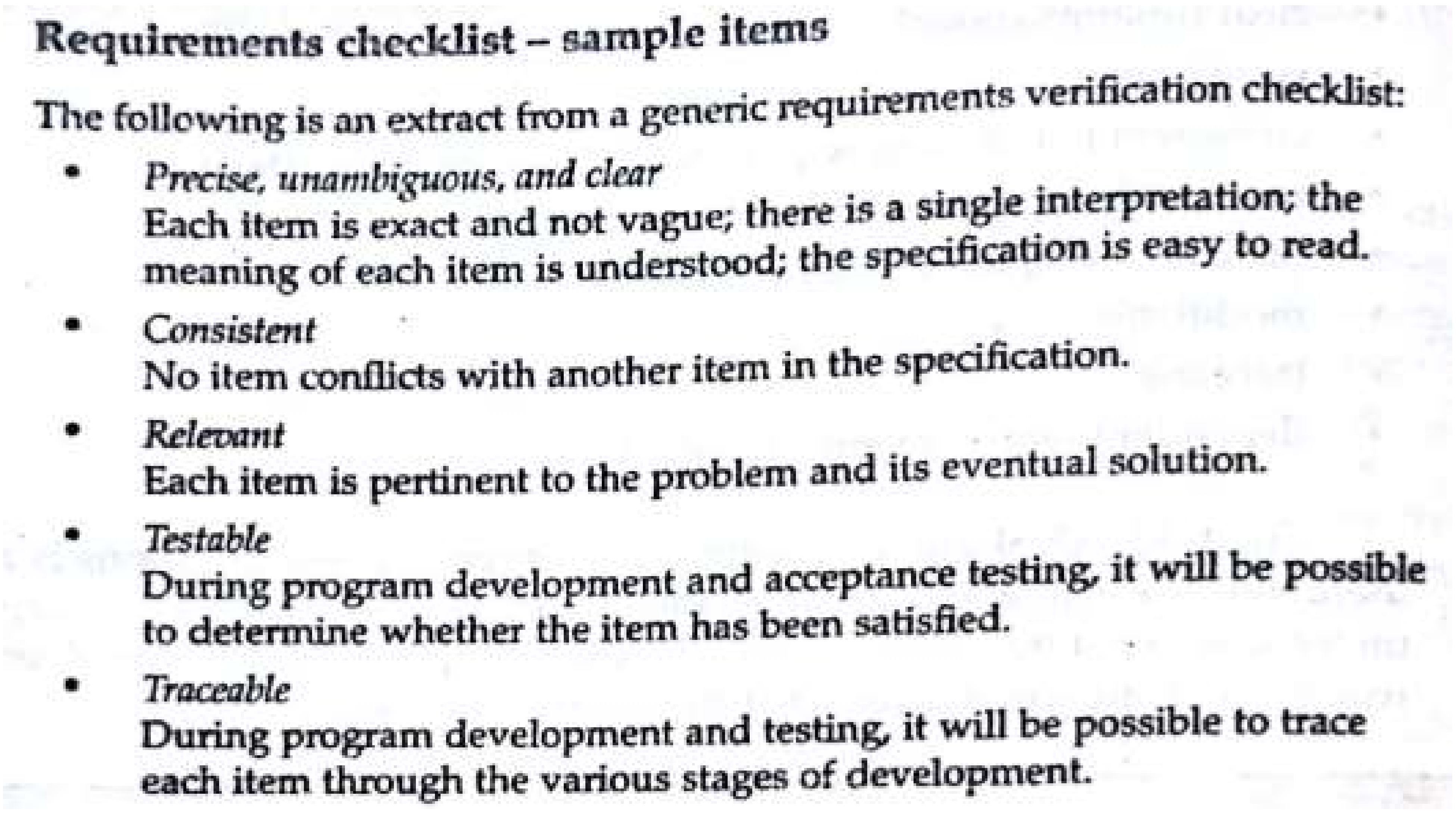
- Any form of human testing, even undisciplined testing.
- Performed other than author,
- Detect defects

Getting leverage on verification
(what and how much verification to do)

- How many defects are we finding
- How many are found during review and later validation.
- What percentage are left at the end and only being found by customers.

Verifying documents at different phases

- Verifying requirements
  – Understand users need
  – Capability needed by users
  – What they want.

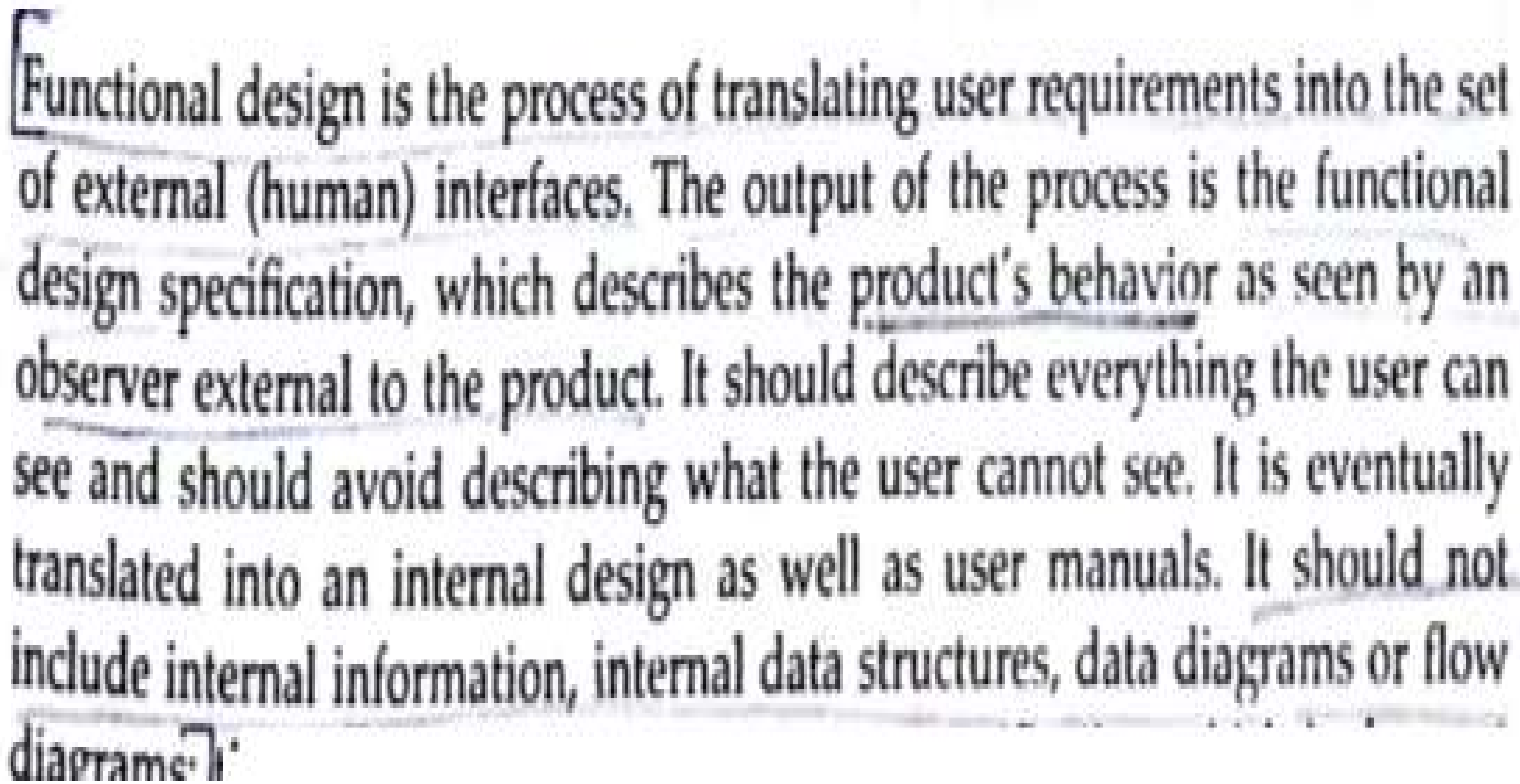


Requirements checklist – sample items

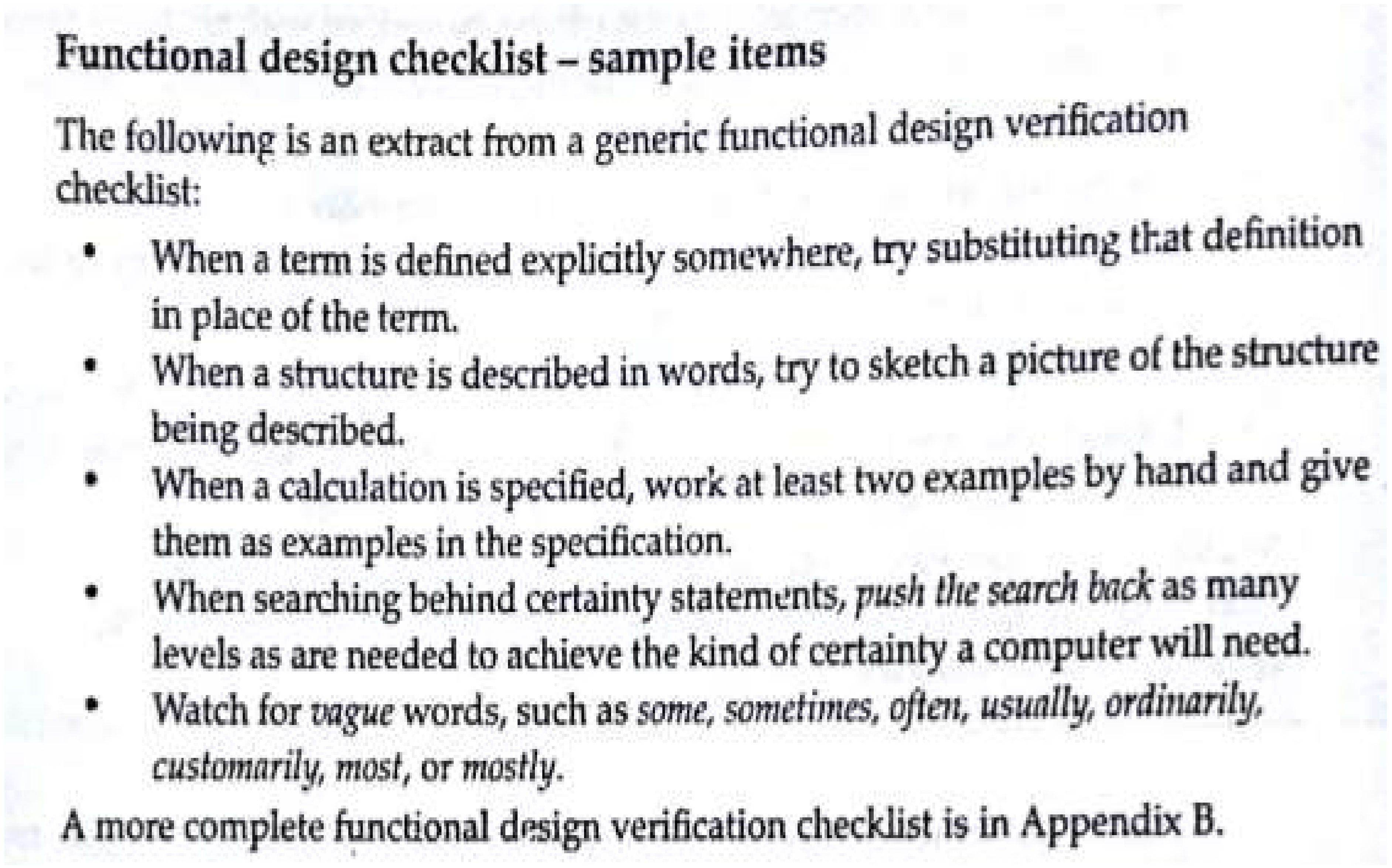
The following is an extract from a generic requirements verification checklist:

- *Precise, unambiguous, and clear*
  Each item is exact and not vague; there is a single interpretation; the meaning of each item is understood; the specification is easy to read.
- *Consistent*
  No item conflicts with another item in the specification.
- *Relevant*
  Each item is pertinent to the problem and its eventual solution.
- *Testable*
  During program development and acceptance testing, it will be possible to determine whether the item has been satisfied.
- *Traceable*
  During program development and testing, it will be possible to trace each item through the various stages of development.

Verifying the functional design



Functional design is the process of translating user requirements into the set of external (human) interfaces. The output of the process is the functional design specification, which describes the product's behavior as seen by an observer external to the product. It should describe everything the user can see and should avoid describing what the user cannot see. It is eventually translated into an internal design as well as user manuals. It should not include internal information, internal data structures, data diagrams or flow diagrams.



Functional design checklist – sample items

The following is an extract from a generic functional design verification checklist:

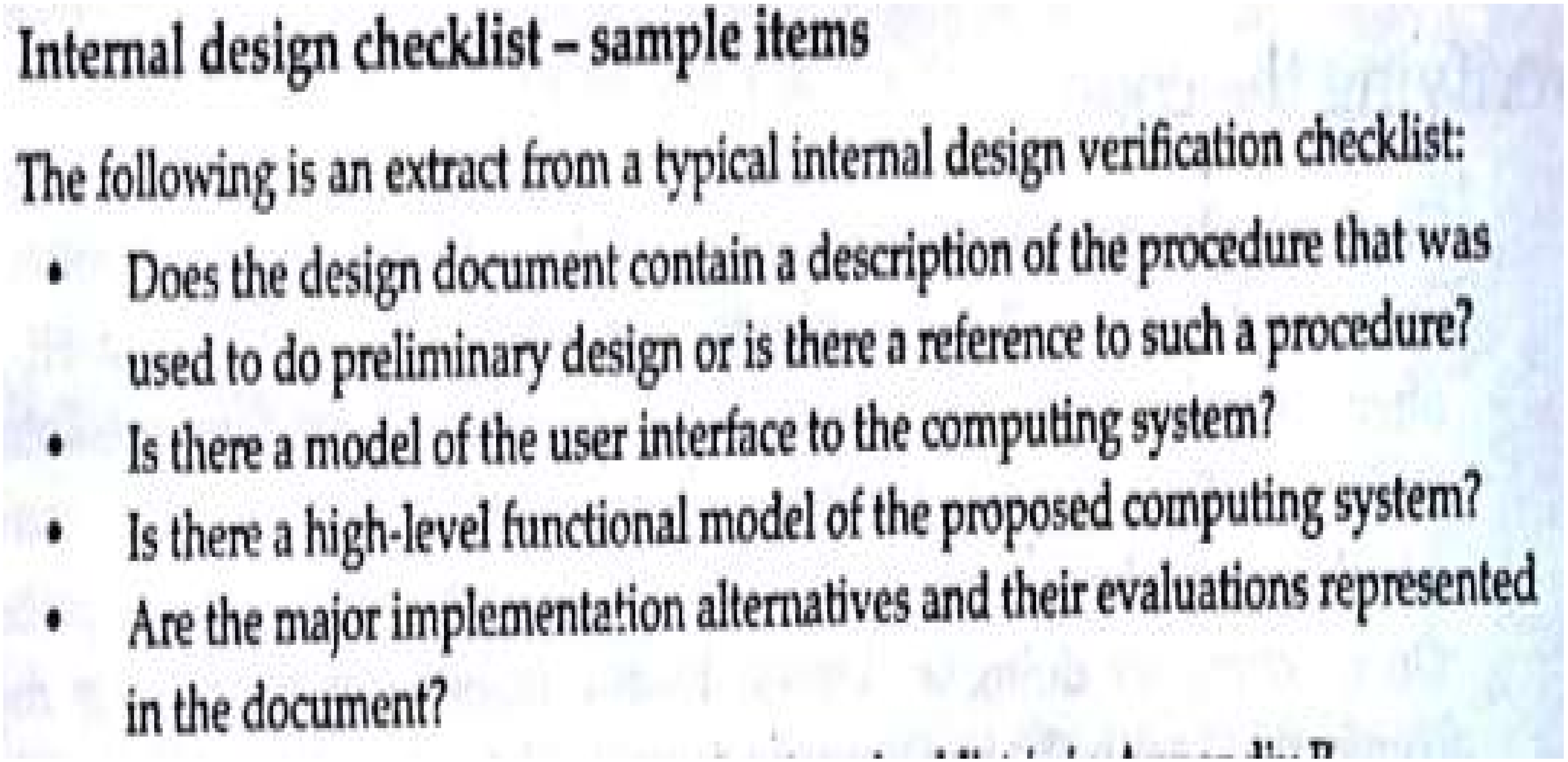
- When a term is defined explicitly somewhere, try substituting that definition in place of the term.
- When a structure is described in words, try to sketch a picture of the structure being described.
- When a calculation is specified, work at least two examples by hand and give them as examples in the specification.
- When searching behind certainty statements, *push the search back* as many levels as are needed to achieve the kind of certainty a computer will need.
- Watch for *vague* words, such as *some, sometimes, often, usually, ordinarily, customarily, most*, or *mostly*.

A more complete functional design verification checklist is in Appendix B.

- Goal is :
  - How successful user requirements have been incorporated into functional design.
  - failings of FDS is Incompleteness.
  - What’s missing.

Internal design

- Translating functional specification into internal design.
- Data flows, algorithms.
- How the product is built.
- Reviewing internal design involved using checklist tracing the path back



Internal design checklist – sample items

The following is an extract from a typical internal design verification checklist:

- Does the design document contain a description of the procedure that was used to do preliminary design or is there a reference to such a procedure?
- Is there a model of the user interface to the computing system?
- Is there a high-level functional model of the proposed computing system?
- Are the major implementation alternatives and their evaluations represented in the document?

Verifying code

- Coding is a process of translating detailed design specification into specific code.
- Do walkthroughs and inspections on code.

(1) Comparing the code with internal design specifications.
(2) Examining the code against a language-specific checklist.
(3) Using a static analysis tool to check for compliance with the syntactic/content requirements.
(4) Verifying the correspondence of terms in code with data dictionary and with internal design specification.
(5) Searching for new boundary conditions, possible performance bottle-necks, and other internal considerations which may form the basis for additional validation tests.

Code checklist – sample items

The following are typical headings with a single example under each from a generic code verification checklist:

Data reference errors
Is an unset or unitialized variable referenced?

Data declaration errors
Are there variables with similar names?

Computation errors
Is the target variable of an assignment smaller than the right-hand expression?

Comparison errors
Are the conversion rules for comparisons between data or variables of inconsistent type or length handled?

Control flow errors
Is there a possibility of premature loop exit?

Interface errors
If the module has multiple entry points, is a parameter ever referenced that is not associated with the current point of entry?

Input/output errors
Are there grammatical errors in program output text?

Portability
How is the data organized (e.g., packed structures)?

Getting best from verification (explain each one)

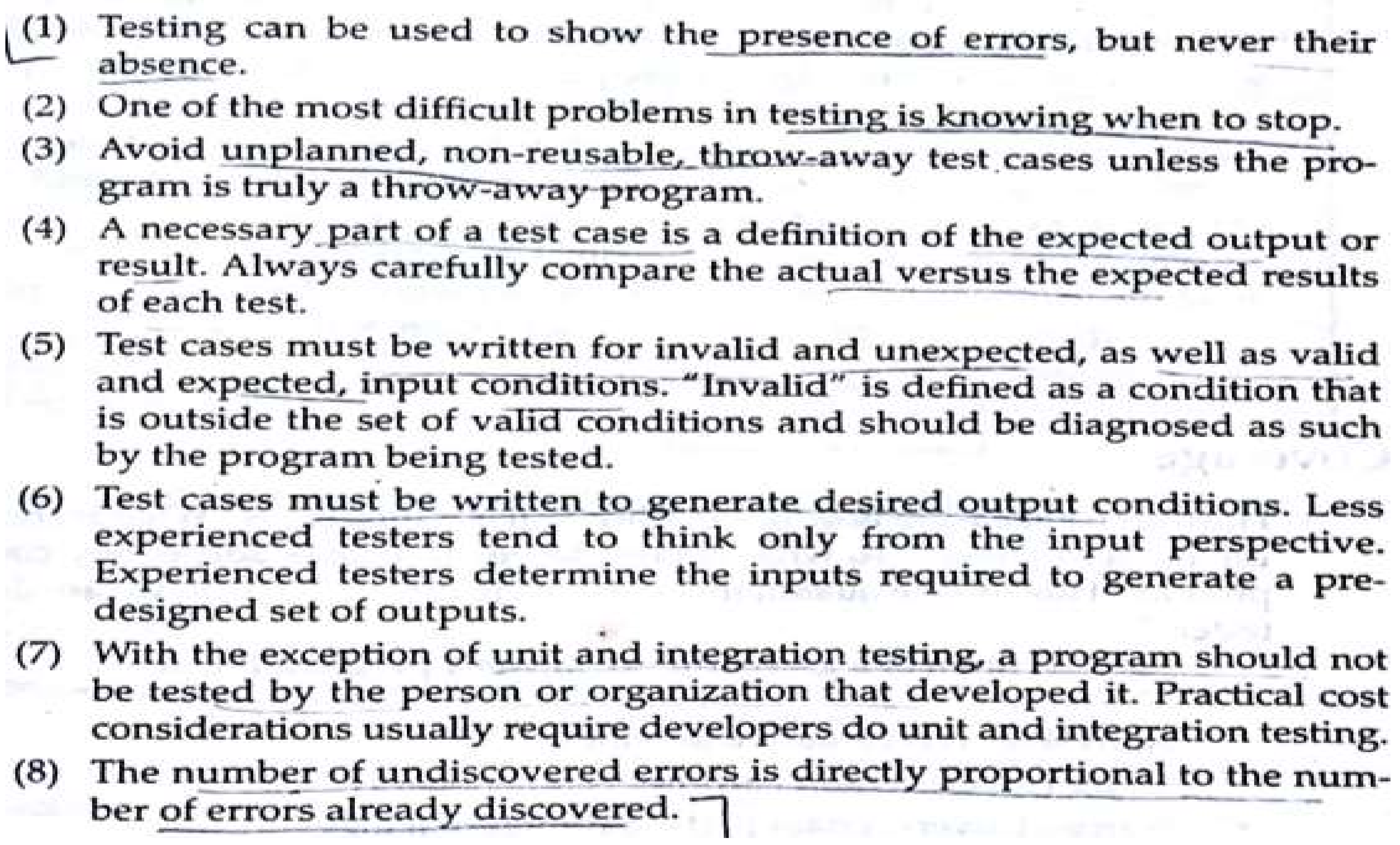
- Author
- Development team
- Inspection team
- Cost effective verification

Three critical success factors for implementing verification

- Success factor 1 : Process ownership
- Success factor 2:Management support
- Success factor 3:Training

Validation testing

8 axioms that apply to all validation testing



(1) Testing can be used to show the presence of errors, but never their absence.

(2) One of the most difficult problems in testing is knowing when to stop.

(3) Avoid unplanned, non-reusable, throw-away test cases unless the program is truly a throw-away program.

(4) A necessary part of a test case is a definition of the expected output or result. Always carefully compare the actual versus the expected results of each test.

(5) Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions. "Invalid" is defined as a condition that is outside the set of valid conditions and should be diagnosed as such by the program being tested.

(6) Test cases must be written to generate desired output conditions. Less experienced testers tend to think only from the input perspective. Experienced testers determine the inputs required to generate a predesigned set of outputs.

(7) With the exception of unit and integration testing, a program should not be tested by the person or organization that developed it. Practical cost considerations usually require developers do unit and integration testing.

(8) The number of undiscovered errors is directly proportional to the number of errors already discovered.

The IEEE/ANSI definition is as follows:

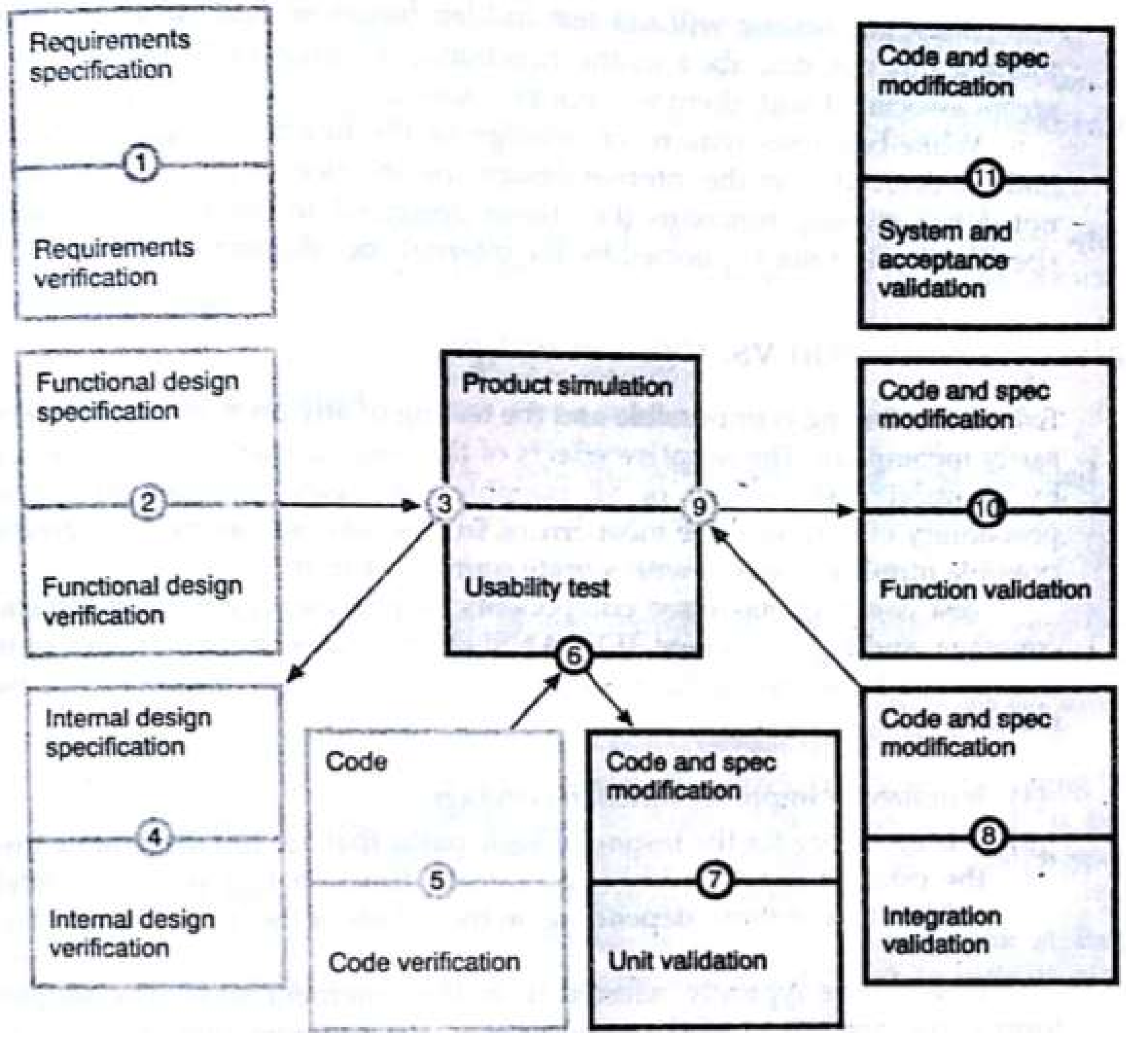
Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Coverage

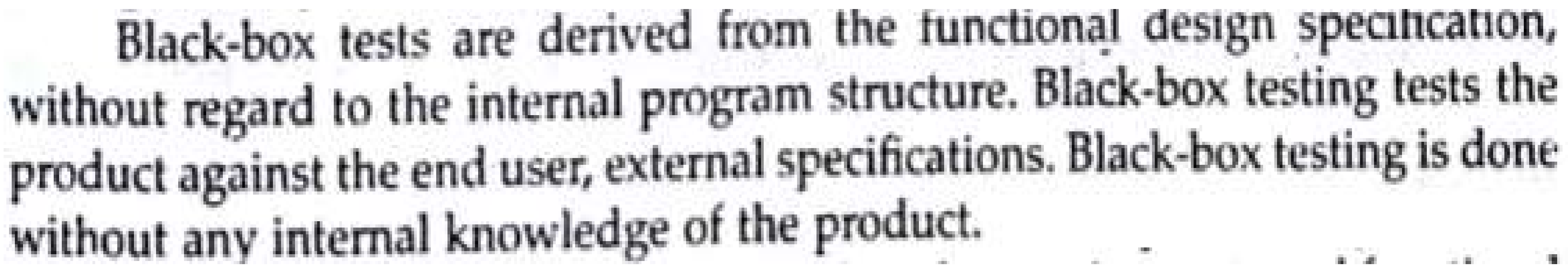
How do we measure how thoroughly tested a product is? What is the measure of "testedness"? To what degree do our test cases adequately cover the product? How do we quantitatively measure how good a job we are doing as testers?

The execution of a given test case against program P will:

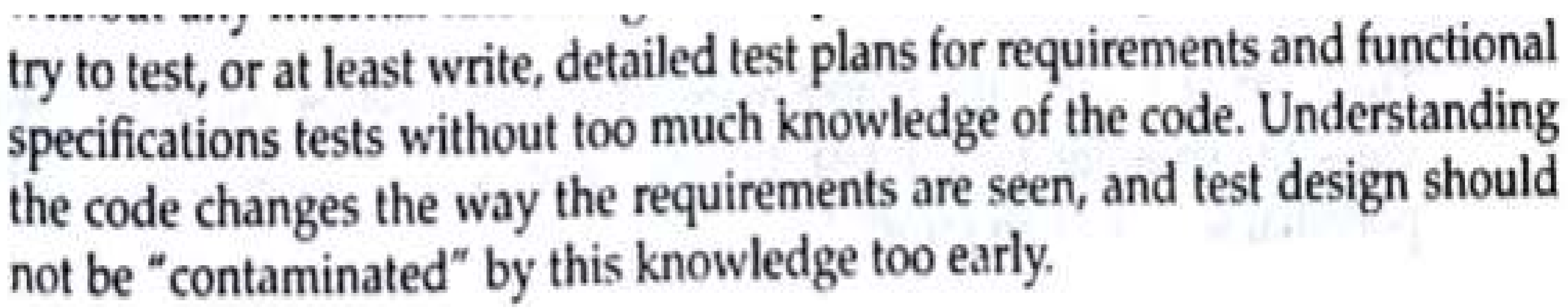
- address (cover) certain requirements of P;
- utilize (cover) certain parts of P's functionality;
- exercise (cover) certain parts of P's internal logic.


Requirements specification
1
Requirements verification
Functional design specification
2
Functional design verification
Internal design specification
4
Internal design verification
Code
5
Code verification
Product simulation
3
9
Usability test
6
Code and spec modification
7
Unit validation
Code and spec modification
8
Integration validation
Code and spec modification
10
Function validation
Code and spec modification
11
System and acceptance validation

Fundamental testing strategies



Black-box tests are derived from the functional design specification, without regard to the internal program structure. Black-box testing tests the product against the end user, external specifications. Black-box testing is done without any internal knowledge of the product.



try to test, or at least write, detailed test plans for requirements and functional specifications tests without too much knowledge of the code. Understanding the code changes the way the requirements are seen, and test design should not be "contaminated" by this knowledge too early.

Validation mission vs test coverage

- 3 components:
- REQ coverage
- Function coverage
- Logic coverage
- Coverage refers to statement level
- % of statements in the program being executed.

Test basis

- Requirement based
- Functional based
- Internal based

A *test*:

(i) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

(ii) A set of one or more test cases.

IEEE/ANSI's second definition of "test" requires additional terms to identify separate and distinct, lower-level tests within a single test case.

A *test case*:

(i) A set of test inputs, execution conditions, and expected results developed for a particular objective.

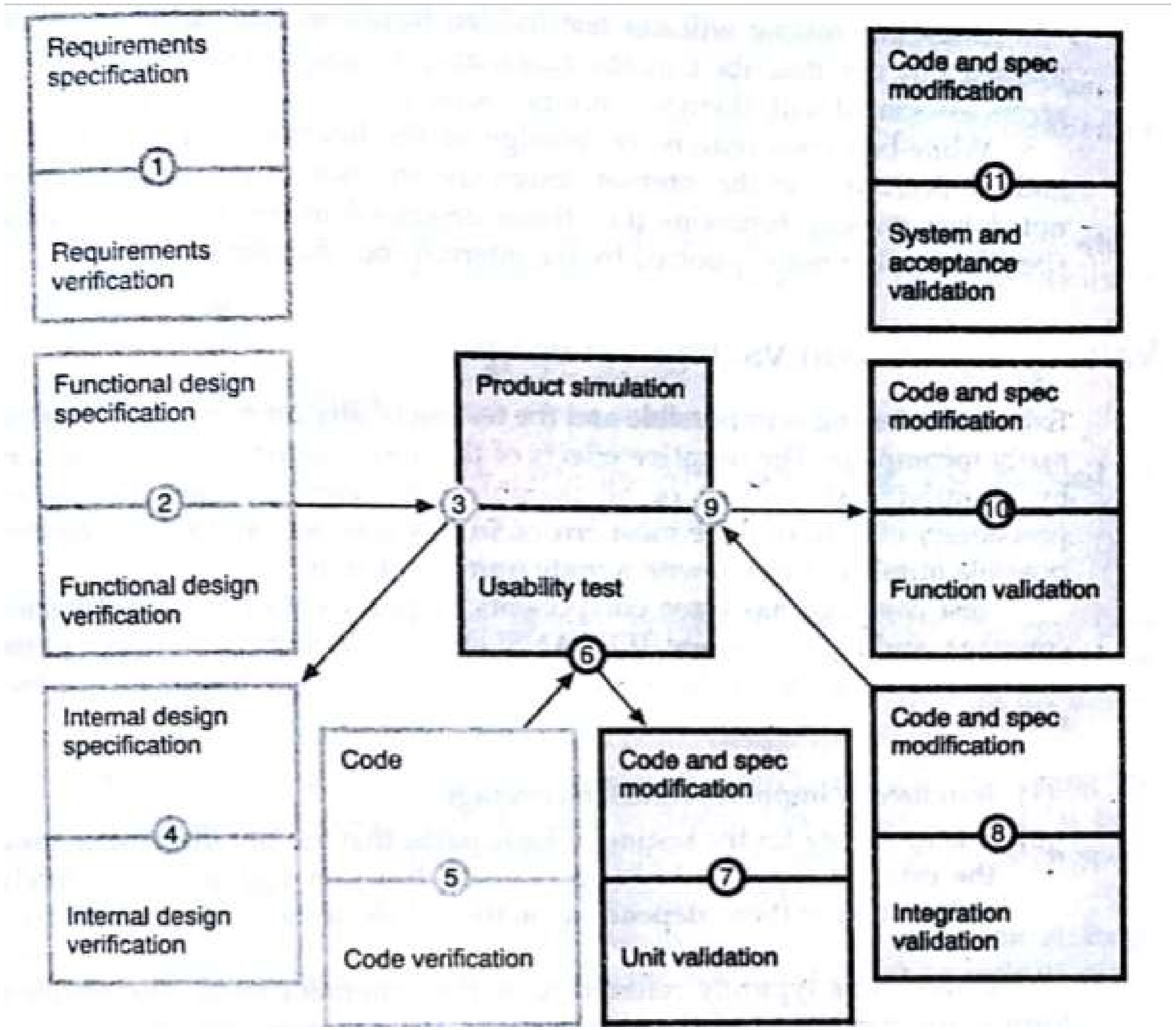
(ii) The smallest entity that is always executed as a unit, from beginning to end.

A test case may perform any number of discrete *subtests*.

A *test procedure*:

(i) The detailed instructions for the set-up, execution, and evaluation of results for a given test case.

(ii) A test case may be used in more than one test procedure. (See Figure 8.2.)

- 
Requirements specification
1
Requirements verification
Functional design specification
2
Functional design verification
Internal design specification
4
Internal design verification
Code
5
Code verification
Product simulation
3
9
Usability test
6
Code and spec modification
7
Unit validation
Code and spec modification
8
Integration validation
Code and spec modification
10
Function validation
Code and spec modification
11
System and acceptance validation

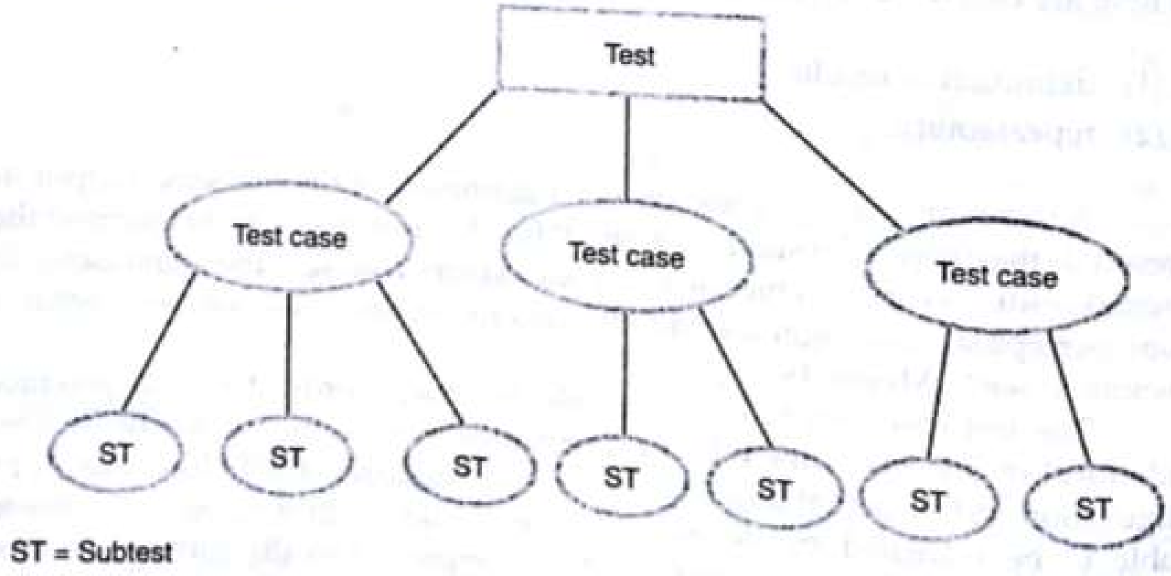
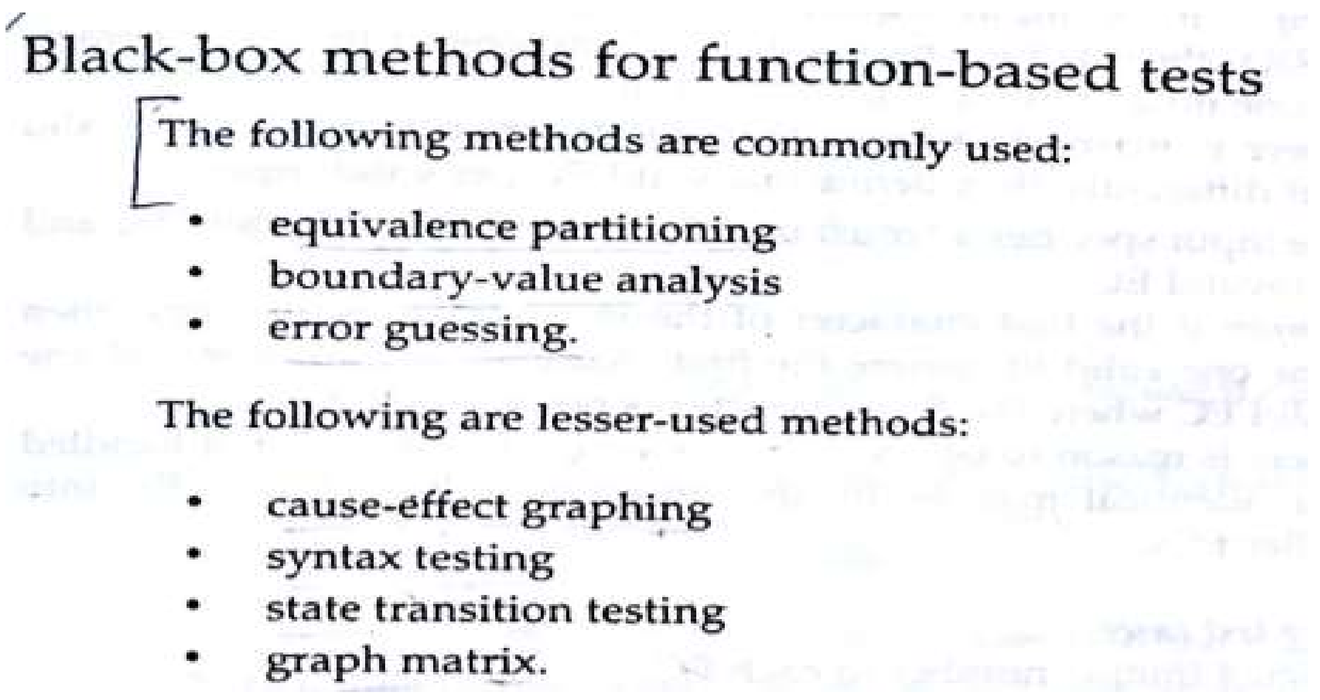


Figure 8.2 Tests, test cases, and subtests. (© 1993, 1994 Software Development Technologies)

- 

Black-box methods for function-based tests

The following methods are commonly used:

- equivalence partitioning
- boundary-value analysis
- error guessing.

The following are lesser-used methods:

- cause-effect graphing
- syntax testing
- state transition testing
- graph matrix.

Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

Equivalence partitioning

- Equivalence partitioning is a software testing technique that divides the input and/or output data of a software unit into partitions of data from which test cases can be derived.
- The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object.
- Test cases are designed to cover each partition at least once.

What can be found using equivalence partitioning?

- Equivalence partitioning technique uncovers classes of errors.

- Testing uncovers sets of inputs that causes errors or failures, not just individual inputs.

What can be partitioned?

- Usually it is the input data that is partitioned.
- However, depending on the software unit to be tested, output data can be partitioned as well.
- Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered ‘equivalent’).

Recommendations on defining partitions

A number of items must be considered:

- All valid input data for a given condition are likely to go through the same process.
- Invalid data can go through various processes and need to be evaluated more carefully. For example:
  - a blank entry may be treated differently than an incorrect entry,
  - a value that is less than a range of values may be treated differently than a value that is greater,
  - if there is more than one error condition within a particular function, one error may override the other, which means the subordinate error does not get tested unless the other value is valid.

Equivalence partitioning

- Equivalence partitioning is a software testing technique that divides the input and/or output data of a software unit into partitions of data from which test cases can be derived.
- The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object.
- Test cases are designed to cover each partition at least once.

Equivalence partitioning

(Equivalence partitioning is a systematic process that identifies, on the basis of whatever information is available, a set of interesting classes of input conditions to be tested) where each class is representative of (or covers) a large set of other possible tests. (If partitioning is applied to the product under test, the product is going to behave in much the same way for all members of the class.)

What can be found using equivalence partitioning?

- Equivalence partitioning technique uncovers classes of errors.

- Testing uncovers sets of inputs that causes errors or failures, not just individual inputs.

What can be partitioned?

- Usually it is the input data that is partitioned.
- However, depending on the software unit to be tested, output data can be partitioned as well.
- Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered ‘equivalent’).

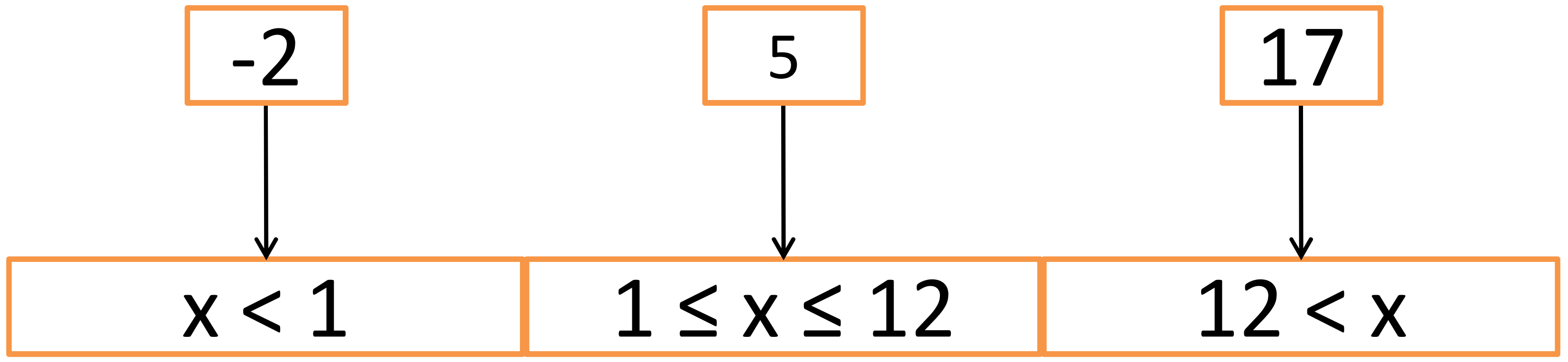
Equivalence partitioning example

- Example of a function which takes a parameter “month”.
- The valid range for the month is 1 to 12, representing January to December. This valid range is called a partition.
- In this example there are two further partitions of invalid ranges.

$x < 1$	$1 \le x \le 12$	$12 < x$

Equivalence partitioning example

- Test cases are chosen so that each partition would be tested.



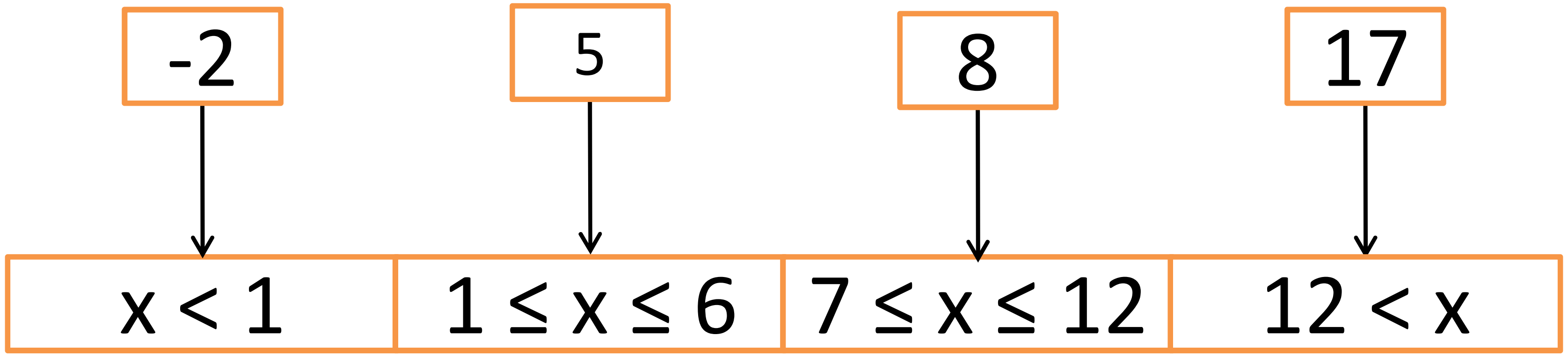
So, is it black box?

- The tendency is to relate equivalence partitioning to so called ***black box*** testing.

- However ***grey box*** technique might be applied as well.

Equivalence partitioning using grey-box technique

- Depending upon the input the software internally will run through ***different paths*** to perform slightly different actions according to the part of the year.
- Therefore middle partition is divided into two subpartitions.



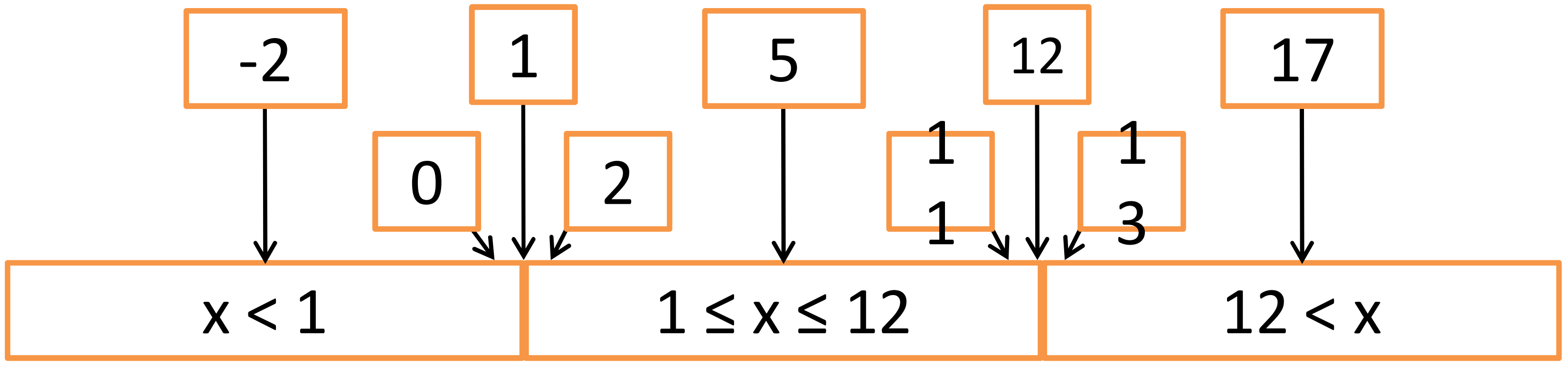
Boundary value analysis

- Equivalence partitioning is not a stand alone method to determine test cases. It is usually supplemented by ***boundary value analysis***.

- ***Boundary value analysis*** focuses on values on the edge of an equivalence partition or at the smallest value on either side of an edge.

Equivalence partitioning with boundary value analysis

- We use the same example as before.
- Test cases are supplemented with ***boundary values***.



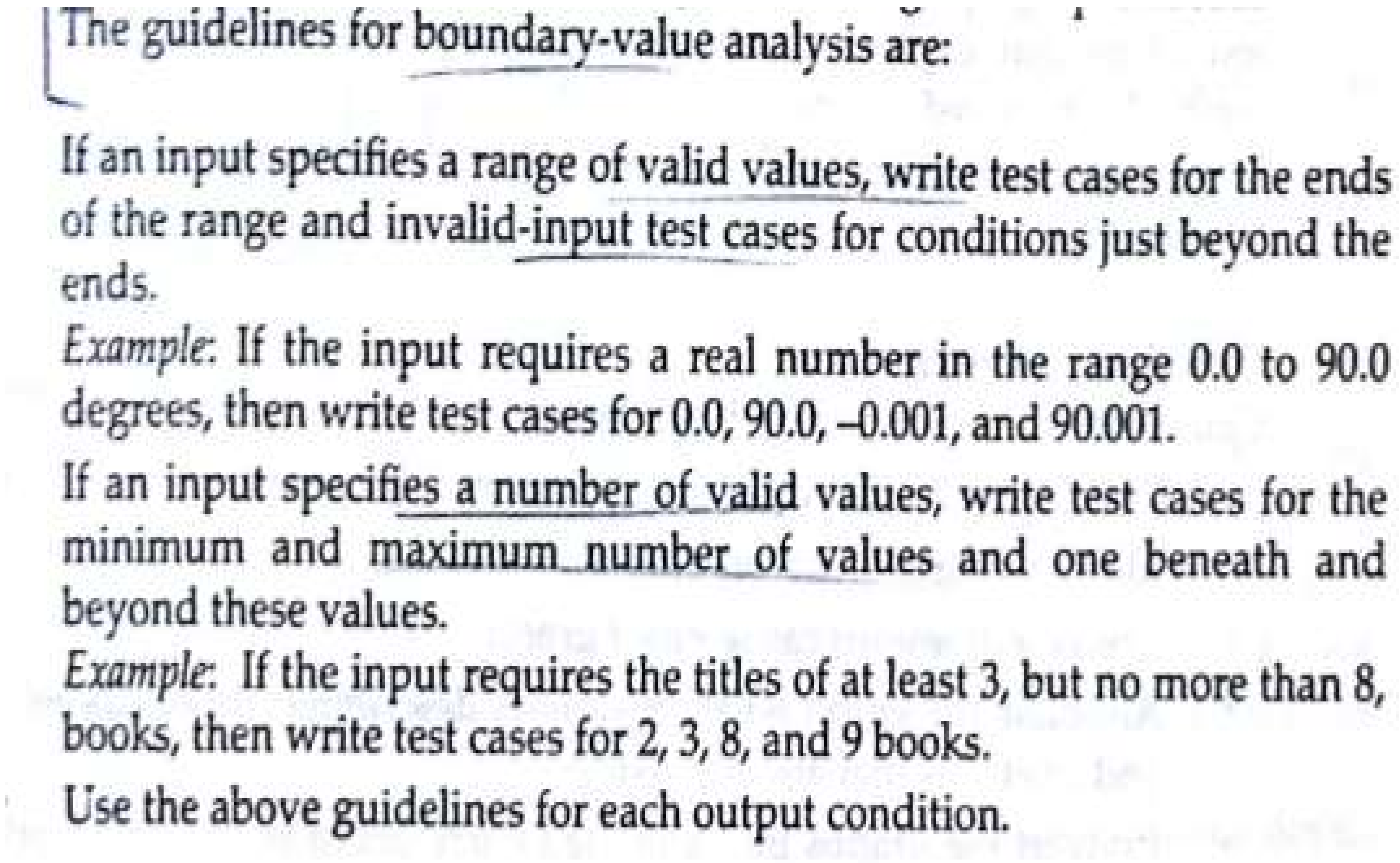
Summary

Pros:

- optimum test case size, therefore time-saving;
- uncovers a class of error, not just an error with specific data input.

Cons:

- possible mistakes when defining partitions;
- does not test all inputs.



The guidelines for boundary-value analysis are:

- If an input specifies a range of valid values, write test cases for the ends of the range and invalid-input test cases for conditions just beyond the ends.
  Example: If the input requires a real number in the range 0.0 to 90.0 degrees, then write test cases for 0.0, 90.0, –0.001, and 90.001.
  If an input specifies a number of valid values, write test cases for the minimum and maximum number of values and one beneath and beyond these values.
  Example: If the input requires the titles of at least 3, but no more than 8, books, then write test cases for 2, 3, 8, and 9 books.
  Use the above guidelines for each output condition.

Boundary analyze and EP

Example:

- " if a pupil has total score >= 75, he will pass the exam, otherwise will fail (total score is an integer)"

Conditions	Valid equivalence classes	Invalid equivalence classes
Total score	1. >=75	2. <75 3. Null 4. String
Result of the exam	5. Pass 6. Fail 7. Error message	

→

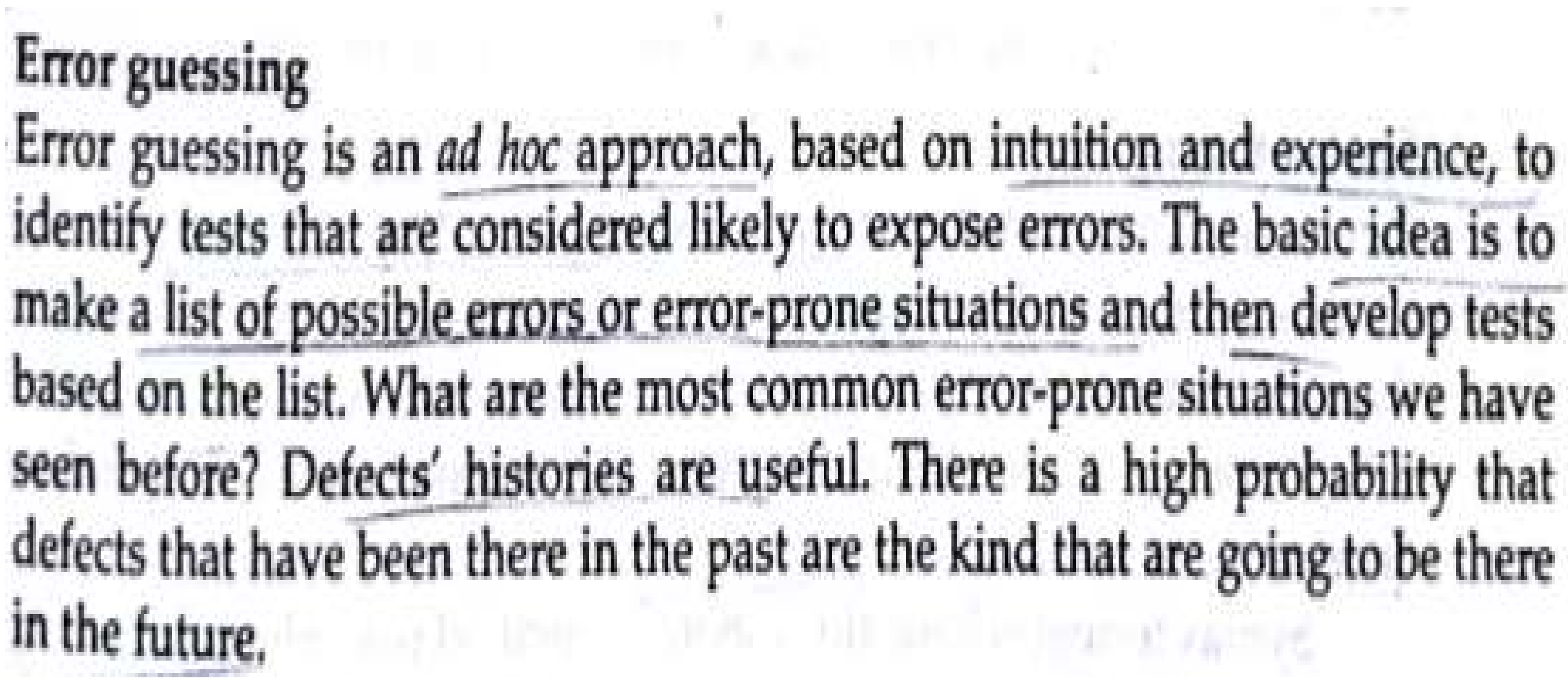
Test case:

1. 1, 5
2. 2, 6
3. 3, 7
4. 4, 7

→

Data to test:

1a. 75, pass
1b. 76, pass
2. 74, fail
3. Null, error message
4a. A, error message
4b. I am a tester of EW and I love this job, error message



Error guessing

Error guessing is an *ad hoc* approach, based on intuition and experience, to identify tests that are considered likely to expose errors. The basic idea is to make a list of possible errors or error-prone situations and then develop tests based on the list. What are the most common error-prone situations we have seen before? Defects' histories are useful. There is a high probability that defects that have been there in the past are the kind that are going to be there in the future.

- Cause Effect Graph is a black box testing technique that graphically illustrates the relationship between a given outcome and all the factors that influence the outcome.

Steps to proceed on Cause-Effect Diagram:

Firstly: Recognize and describe the input conditions (causes) and actions (effect)

Secondly: Build up a cause-effect graph

Third: Convert cause-effect graph into a decision table

Fourth: Convert decision table rules to test cases. Each column of the decision table represents a test case

- Syntax Testing, a black box testing technique, involves testing the System inputs and it is usually automated because syntax testing produces a large number of tests.

Syntax Testing steps:

- Identify the target language or format
- Define the syntax of the language
- Validate and debug the syntax

- **State transition testing** is used where some aspect of the system can be described in what is called a ‘finite state machine’. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the ‘machine’..

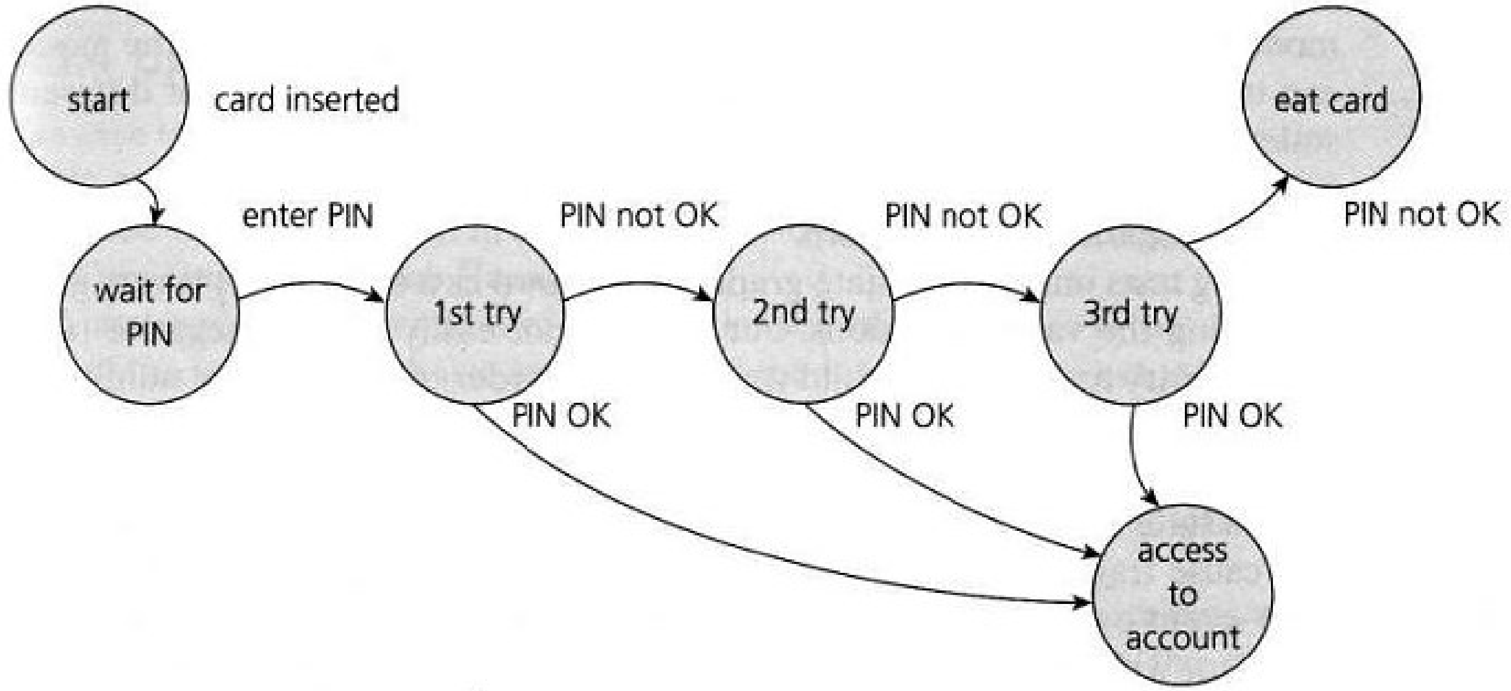
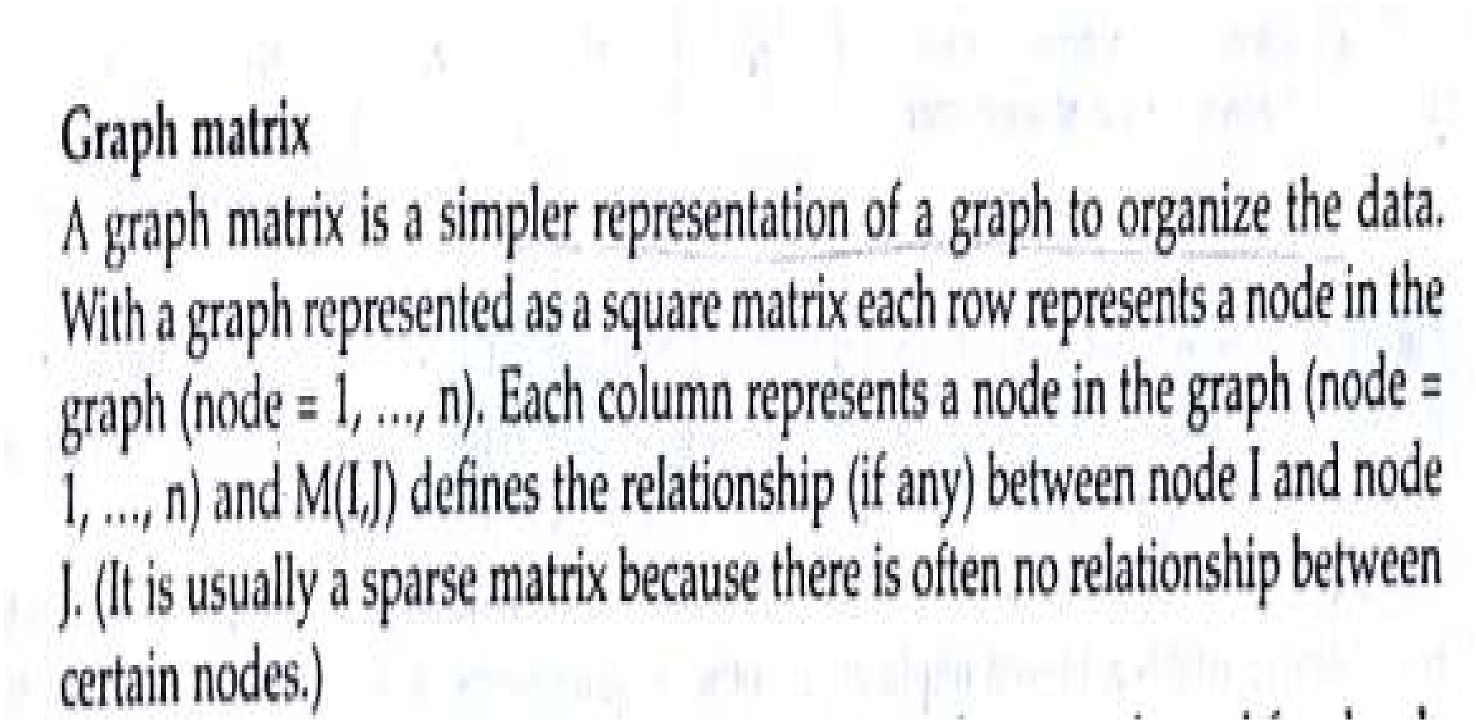


FIGURE 4.2 State diagram for PIN entry



Graph matrix

A graph matrix is a simpler representation of a graph to organize the data. With a graph represented as a square matrix each row represents a node in the graph (node = 1, ..., n). Each column represents a node in the graph (node = 1, ..., n) and M(I,J) defines the relationship (if any) between node I and node J. (It is usually a sparse matrix because there is often no relationship between certain nodes.)

White-box methods for internals-based tests

Once white-box testing is started, there are a number of techniques to ensure the internal parts of the system are being adequately tested and that there is sufficient logic coverage.

The execution of a given test case against program P will exercise (cover) certain parts of P's internal logic. A measure of testedness for P is the degree of logic coverage produced by the collective set of test cases for P. White-box testing methods are used to increase logic coverage.

There are four basic forms of logic coverage:

(1) statement coverage
(2) decision (branch) coverage
(3) condition coverage
(4) path coverage.

		Statement coverage	Decision coverage	Condition coverage	Decision/ condition coverage	Multiple condition coverage
1	Each statement is executed at least once	Y	Y	Y	Y	Y
2	Each decision takes on all possible outcomes at least once	N	Y	N	Y	implicit
3	Each condition in a decision takes on all possible outcomes at least once	N	N	Y	Y	implicit
4	All possible combinations of condition outcomes in each decision occur at least once	N	N	N	N	Y

The three input parameters are age (integer), sex (male or female), and married (true or false). Keep in mind the following:

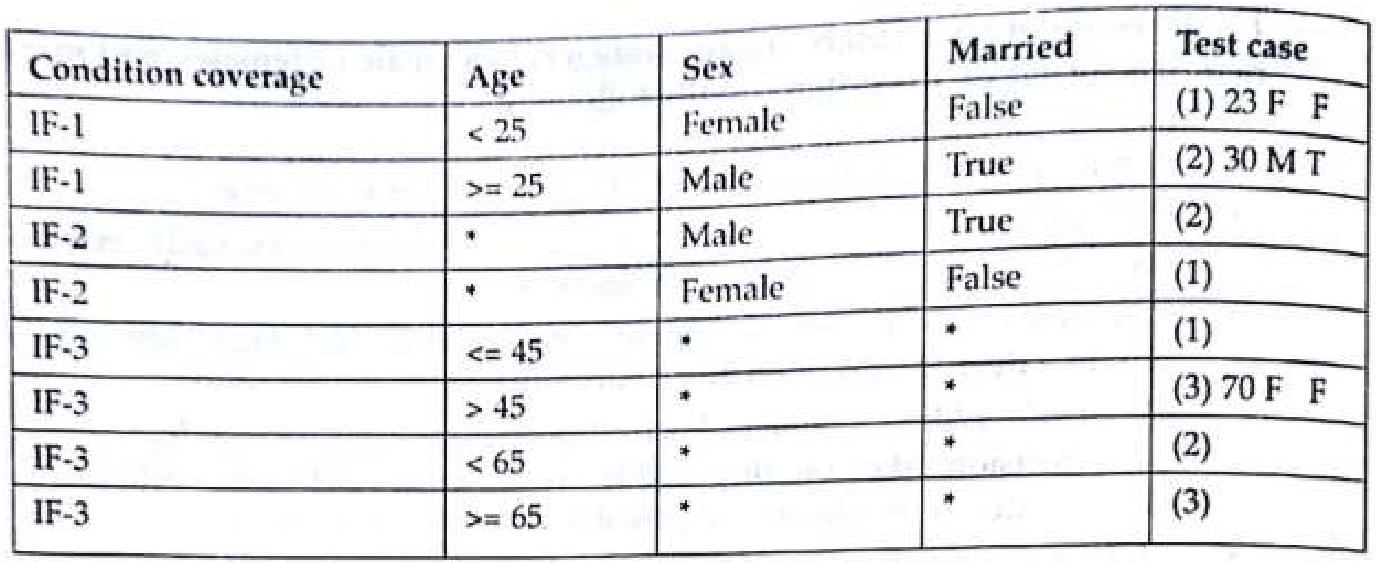
- *Statement coverage*: Each statement is executed at least once.
- *Decision coverage*: Each statement is executed at least once; each decision takes on all possible outcomes at least once.
- *Condition coverage*: Each statement is executed at least once; each condition in a decision takes on all possible outcomes at least once.
- *Decision/condition coverage*: Each statement is executed at least once; each decision takes on all possible outcomes at least once; each condition in a decision takes on all possible outcomes at least once.
- *Multiple/condition coverage*: Each statement is executed at least once; all possible combinations of condition outcomes in each decision occur at least once.

The liability procedure:

```
procedure liability (age, sex, married, premium ) ;
begin
      premium := 500 ;
      if ( (age < 25) and (sex = male) and (not married) ) then premium :=
      premium + 1500 ;
      else ( if ( married or (sex = female) ) then
                        premium := premium - 200 ;
            if ( (age > 45) and (age < 65) ) then
                        premium := premium - 100 ; )
end ;
```

Statement coverage	Age	Sex	Married	Test case

Decision coverage	Age	Sex	Married	Test case
IF-1	< 25	Male	False	(1) 23 M F
IF-1	< 25	Female	False	(2) 23 F F
IF-2	*	Female	*	(2)
IF-2	>= 25	Male	False	(3) 50 M F
IF-3	<= 45	Female (n1)	*	(2)
IF-3	> 45, < 65	*	*	(3)



Condition coverage	Age	Sex	Married	Test case
IF-1	< 25	Female	False	(1) 23 F F
IF-1	>= 25	Male	True	(2) 30 M T
IF-2	*	Male	True	(2)
IF-2	*	Female	False	(1)
IF-3	<= 45	*	*	(1)
IF-3	> 45	*	*	(3) 70 F F
IF-3	< 65	*	*	(2)
IF-3	>= 65	*	*	(3)

Validation activities

- Low level testing
  - Unit
  - integration
- High level testing
  - Usability
  - Function
  - System
  - Acceptance

Unit Testing

- Algorithms and logic
- Data structures (global and local)
- Interfaces
- Independent paths
- Boundary conditions
- Error handling

Unit testing manages the combinations of testing. It facilitates error diagnosis and correction by development and it allows parallelism, in other words, testing multiple components simultaneously.]

Testing a given module (X) in isolation may require:

(1) a driver module which transmits test cases in the form of input arguments to X and either prints or interprets the results produced by X;

(2) zero or more "stub" modules each of which simulates the function of a module called by X. It is required for each module that is directly subordinate to X in the execution hierarchy. If X is a terminal module (i.e., it calls no other modules), then no stubs are required.

Integration Testing

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
  - Non-incremental Integration Testing
  - Incremental Integration Testing

Non-incremental Integration Testing

- Commonly called the “Big Bang” approach
- All components are combined in advance
- The entire program is tested as a whole
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental Integration Testing

- Two kinds
  - Top-down integration
  - Bottom-up integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Why Integration Testing Is Necessary

- One module can have an adverse effect on another
- Subfunctions, when combined, may not produce the desired major function
- Individually acceptable imprecision in calculations may be magnified to unacceptable levels

Why Integration Testing Is Necessary (cont’d)

- Interfacing errors not detected in unit testing may appear
- Timing problems (in real-time systems) are not detectable by unit testing
- Resource contention problems are not detectable by unit testing

The steps in bottom-up integration are:

Begin with the terminal modules (those that do not call other modules) of the hierarchy.

A driver module is produced for every module.

The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested.

After a module has been tested, its driver is replaced by an actual module (the next one to be tested) and its driver.

The steps in top-down integration are:

- Begin with the top module in the execution hierarchy.
- Stub modules are produced, and some may require multiple versions.
- Stubs are often more complicated than they first appear.
- The next module to be tested is any module with at least one previously tested superordinate (calling) module.
- After a module has been tested, one of its stubs is replaced by an actual module (the next one to be tested) and its required stubs.

Top-Down Integration

1. The main control module is used as a driver, and stubs are substituted for all modules directly subordinate to the main module.
2. Depending on the integration approach selected (depth or breadth first), subordinate stubs are replaced by modules one at a time.

Top-Down Integration (cont’d)

3. Tests are run as each individual module is integrated.
4. On the successful completion of a set of tests, another stub is replaced with a real module
5. Regression testing is performed to ensure that errors have not developed as result of integrating new modules

Problems with Top-Down Integration

- Many times, calculations are performed in the modules at the bottom of the hierarchy
- Stubs typically do not pass data up to the higher modules
- Delaying testing until lower-level modules are ready usually results in integrating many modules at the same time rather than one at a time
- Developing stubs that can pass data up is almost as much work as developing the actual module

Bottom-Up Integration

- Integration begins with the lowest-level modules, which are combined into clusters, or builds, that perform a specific software subfunction
- Drivers (control programs developed as stubs) are written to coordinate test case input and output
- The cluster is tested
- Drivers are removed and clusters are combined moving upward in the program structure

Problems with Bottom-Up Integration

- The whole program does not exist until the last module is integrated
- Timing and resource contention problems are not found until late in the process

- Unit testing
  - Concentrates on each component/function of the software as implemented in the source code.
- System testing
  - The software and other system elements are tested as a whole
- Alpha testing
  - Conducted at the developer’s site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- Beta testing
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals

Validation Testing

- Determine if the software meets all of the requirements defined in the SRS
- Having written requirements is essential
- Regression testing is performed to determine if the software still meets all of its requirements in light of changes and modifications to the software
- Regression testing involves selectively repeating existing validation tests, not developing new tests

Alpha and Beta Testing

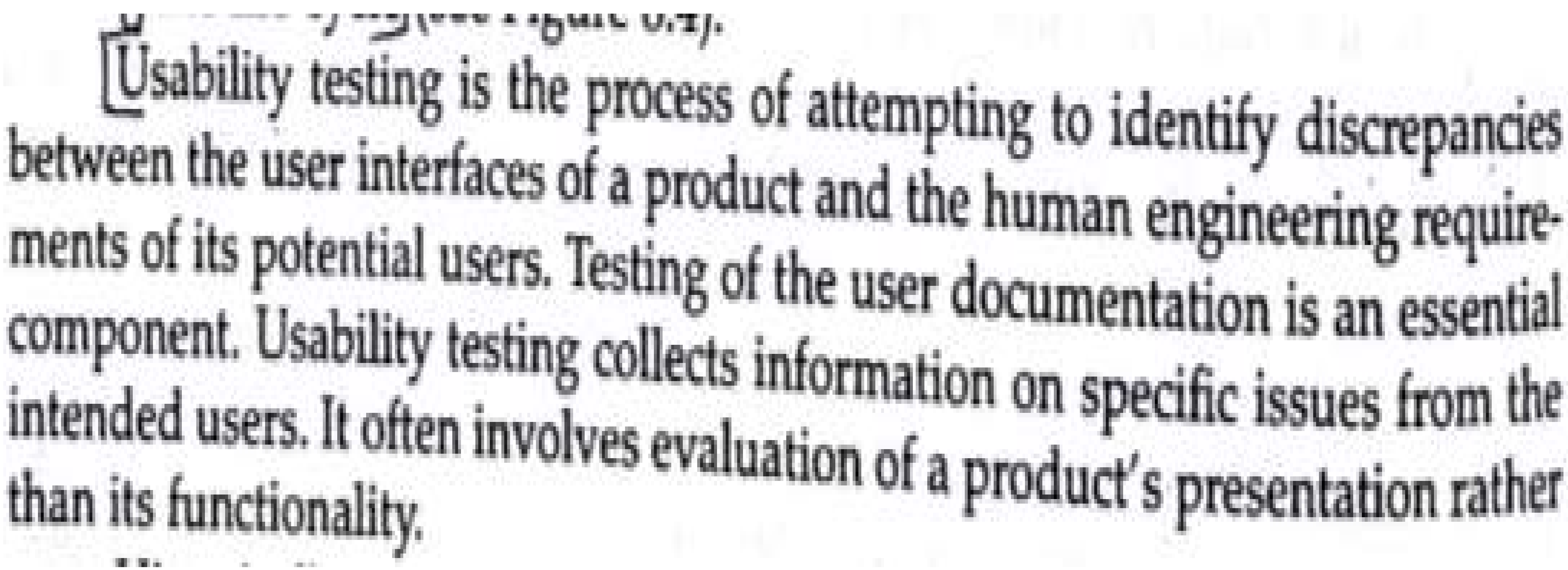
- It’s best to provide customers with an outline of the things that you would like them to focus on and specific test scenarios for them to execute.
- Provide with customers who are actively involved with a commitment to fix defects that they discover.

Acceptance Testing

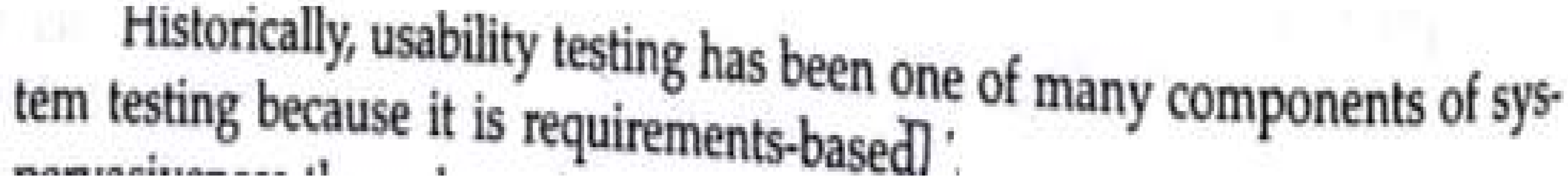
- Similar to validation testing except that customers are present or directly involved.
- Usually the tests are developed by the customer

High level testing

- Usability
- Function testing
- System testing
- Acceptance testing



[Usability testing is the process of attempting to identify discrepancies between the user interfaces of a product and the human engineering requirements of its potential users. Testing of the user documentation is an essential component. Usability testing collects information on specific issues from the intended users. It often involves evaluation of a product's presentation rather than its functionality.



Historically, usability testing has been one of many components of system testing because it is requirements-based]